Linux Device Drivers – synchronization and race condition

Jernej Vičič
Introduction

- concurrency,
- what happens when the system is trying to do more than one thing at a time,
- control the concurrency issues is one of the basic problems of operating systems.
Problem example *scull*

- part of the driver code *scull*,
- in this case we present one of the basic problems.
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
    if (!dptr->data[s_pos])
        goto out;
}

- function that writes to a quantum \textit{write},
- decision if the quantum for the next location is already allocated,
two processes: \( A \) and \( B \) try to write data to the same offset from the initial location,
both come to the test \( if \),
if the cursor is NULL, they both try to allocate new memory and get a new pointer to this memory,
both try to allocate,
both succeed in allocating,
only the last one is successful, the first pointer will be overwritten by the content of the last,
this is a classic example \( race \) \( condition \).
- semaphore function $P$ is called $\text{down}$,
- semaphore function $V$ is called $\text{up}$,
/*****************************************/
<asm/semaphore.h>
/*****************************************/
struct semaphore;

/*****************************************/
void sema_init(struct semaphore *sem, int val);

- \textit{val} – starting value
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);

- *name* – semaphore name;
- *DECLARE_MUTEX* – semaphore name is set to 1 (open);
- *DECLARE_MUTEX_LOCKED* – semaphore name is set to 0, closed,
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);

- versions of the `down` function – \( P \);
- `down` – Will keep waiting for the semaphore unless it does not become available. Can not be interrupted.
- `down_interruptible` – Will keep waiting for the semaphore unless it does not become available but can be interrupted. This is always preferable over `down()`.
- `(down_trylock` – Will not put the process to sleep if semaphore is already being held, but returns immediately with non zero return value.
when the thread successfully calls one of the functions *down*, we say that it holds the semaphore (or has successful entered the critical section),

- it can now enter the critical section,
- after the finishing work it calls the function *up*

```c
void up(struct semaphore *sem);
```

- semaphore is released (next process is signalled).
main data structure `scull` is `scull_dev`,
struct scull_dev {
    struct scull_qset *data; /* Pointer to first quantum set */
    int quantum; /* the current quantum size */
    int qset; /* the current array size */
    unsigned long size; /* amount of data stored here */
    unsigned int access_key; /* used by sculluid and scullpriv */
    struct semaphore sem; /* mutual exclusion semaphore */
    struct cdev cdev; /* Char device structure */
};

- *struct semaphore sem,*
- each device has its own semaphore,
- so that processes do not wait for other devices to free the semaphore.
for (i = 0; i < scull_nr_devs; i++) {
    scull_devices[i].quantum = scull_quantum;
    scull_devices[i].qset = scull_qset;
    init_MUTEX(&scull_devices[i].sem);
    scull_setup_cdev(&scull_devices[i], i);
}

- initialize all semaphores (mutex),
- init_MUTEX is called before scull_setup_cdev, that announces the device.
if (down_interruptible(&dev->sem))
    return -ERESTARTSYS;

- put this at the beginning of the `scull_write` function.
- check the return value (it can be terminated by the user -UERESTARTSYS),
out:
    up(&dev->sem);
return retval;

- at the end of the function `scull_write`,
- code releases the semaphore and returns the status (depending on success).